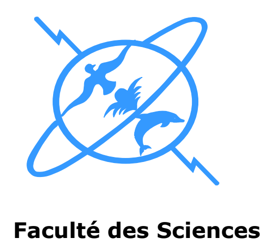
Université Mohammed V-Agdal
Faculté des Sciences Rabat
Département Mathématiques et Informatique
Le module I2 : SMP-SMC, SM-SMI



Le Langage de programmation Maple

Par

Mohamed El Marraki

2006/2007

1. Introduction

L'informatique est la science du traitement automatique de l'information. Pour cela :
1. il faut modéliser cette information,
2. définir à l'aide d'un formalisme strict les traitements dont elle fera l'objet,
3. traduire ces traitements dans un langage compréhensible par un ordinateur.

Les deux points 1 et 2 précédents relèvent de l'*algorithmique*. Le 3ème point concerne ce que l'on nomme la *programmation*.

1.1 L'algorithmique

L'algorithmique est un terme d'origine arabe, hommage à Al Khawarizmi
- Une recette de cuisine est un algorithme!
- Le mode d'emploi d'un magnétoscope est aussi un algorithme!
- indiqué un chemin à un touriste égaré ou faire chercher un objet à quelqu'un par téléphone c'est fabriquer – et faire exécuter – des algorithmes.

Un algorithme, c'est une suite d'instructions, qui une fois exécutée correctement, conduit à un résultat donné.
- Si l'algorithme est juste, le résultat est le résultat voulu, et le touriste se retrouve là où il voulait aller.
- Si l'algorithme est faux, le résultat est, disons, aléatoire, et décidément, ce magnétoscope ne marche pas!!

Pour fonctionner, un algorithme doit donc contenir uniquement des instructions compréhensibles par l'ordinateur.

Remarque :

L'ADN, qui est en quelque sorte le programme génétique, l'algorithme à la base de construction des êtres vivants, est une chaîne construite à partir de quatre éléments invariables. Ce n'est que le nombre de ces éléments, et l'ordre dans lequel ils sont arrangés, qui vont déterminer si on obtient une puce ou un éléphant.

Les ordinateurs eux-mêmes ne sont fondamentalement capables d'exécuter que quatre opérations logiques :
- l'affectation de variables
- la lecture / écriture
- les tests
- les boucles

Un algorithme informatique se ramène donc toujours au bout du compte à la combinaison de ces quatre petites briques de base. Il peut y en avoir quelques unes, quelques dizaines, et jusqu'à plusieurs centaines de milliers dans certains programmes.
La taille d'un algorithme ne conditionne pas en soi sa complexité : de longs algorithmes peuvent être finalement assez simples, et de petits algorithmes peuvent être très compliqués.

1.2 La programmation

Un ordinateur est une machine construite pour exécuter de manière séquentielle des fonctions primitives telles que les calculs arithmétique, les affectations de variables, des testes, des boucles des entrée-sortie, etc. On obtient des traitements plus complexes (comme l'extraction

d'une racine carrée par exemple) en combinant ces opérations entre elles dans un ordre convenable. Comme la vitesse de l'ordinateur est très grande, il n'est pas possible de lui donner des ordres au fur et à mesure de l'avancement du travail, comme on le fait avec une calculette : il faut d'abord dresser la liste des instructions auxquelles la machine devra obéir, dans l'ordre de leur exécution. C'est ce qu'on appelle un **programme**. On charge ensuite ce programme dans la mémoire de la machine, où elle puisera les instructions au fur et à mesure de leur exécution, à sa propre vitesse.

Un programme est la traduction d'un algorithme dans un langage de programmation.
La première difficulté sera d'adapter notre raisonnement humain à la logique de fonctionnement de l'ordinateur. Il faudra être suffisamment précis pour que la machine puisse exécuter la tâche.

Exemple : on veut échanger le contenu de deux cellules mémoires qu'on appellera A et B. La première idée qui consiste à dire "je mets le contenu de A dans B et celui de B dans A" aboutirai à ce que A et B contiendraient les mêmes valeurs, ce n'est pas le résultat souhaité. Le problème est donc un peu plus complexe puisqu'il il faut d'abord sauvegarder le contenu de B dans une cellule C, recopier le contenu de A dans celui de B puis restaurer le contenu de C dans la cellule A.

1.3 Les langages programmation

On classe les langages de programmation en "générations". La première génération regroupe les langages machine, le seconde les langages d'assemblages et la troisième les langages évolués.

1.3.1 Les langages de bas niveau

Le langage machine : Les instructions stockées dans la mémoire de l'ordinateur sont représentées sous la forme de chaînes de chiffres binaires, elles sont exécutées par l'unité de traitement. On dit qu'elles sont exprimées en langage machine. Le langage machine constitue le seul langage réellement "compris" par l'ordinateur.

Exemple :

0	1	0	1	1	0	1	0	0	0	1	1	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1

- Le premier octet (8 bits) signifie le nombre 90 = $5A_{16}$ (hexadécimal) qui est le code opératoire de l'addition à un registre,
- les quatre bits suivants signifient le nombre 3 qui est le numéro de registre,
- les 20 bits restant signifient 517 = 205_{16} (hexadécimal) qui l'adresse de l'opérande.

Cette instruction en langage machine signifie "additionner au contenu du registre 3 le contenu du mot d'adresse 517".

Cette programmation, lente et fastidieuse, était source de nombreuses erreurs. Elle n'est utilisée aujourd'hui que dans quelques cas bien particuliers.

Les langages d'assemblage ont permis l'écriture des instructions du langage machine sous forme symbolique ; la traduction en binaire est assurée par un programme, fourni par le constructeur, appelé assembleur.

Les langages d'assemblage sont des langages de programmation les plus proche du langage machine, où chaque instruction correspond à une instruction machine unique. Toutefois un tel langage est tellement proche du langage machine qu'il dépend étroitement du type de processeur utilisé (chaque type de processeur peut avoir son propre langage machine). Ainsi un programme développé pour une machine ne pourra pas être *porté* sur un autre type de machine (on désigne par le terme "**portable**" un programme qui peut être utilisé sur un grand nombre de machines). Pour pouvoir l'utiliser sur une autre machine il faudra alors parfois réécrire entièrement le programme!
Bien qu'ils introduisent certains allégements ils ne résolvent pas le problème de portabilité (un programme écrit sur une machine ne marche pas sur une autre machine, le programme dépend du processeur).
Cependant les langages d'assemblage sont encore utilisés aujourd'hui dans quelques cas :

- Quand la vitesse d'exécution est primordiale,
- Pour accéder directement à certains périphériques,
- Pour étudier les différentes types d'architecture des ordinateurs.

Pour remédier aux inconvénients des langages d'assemblage, on a crée des langages dites évolués, écrits avec l'alphabet usuel proche du langage humain.

1.3.2 Les langages de haut niveau

L'assembleur est le premier langage informatique qui ait été utilisé. Celui-ci est encore très proche du langage machine mais il permet déjà d'être plus compréhensible. Toutefois un tel langage est tellement proche du langage machine qu'il dépend étroitement du type de processeur utilisé (chaque type de processeur peut avoir son propre langage machine). Ainsi un programme développé pour une machine ne pourra pas être *porté* sur un autre type de machine (on désigne par le terme "**portable**" un programme qui peut être utilisé sur un grand nombre de machines). Pour pouvoir l'utiliser sur une autre machine il faudra alors parfois réécrire entièrement le programme!
Un langage informatique a donc plusieurs avantages:

- il est plus facilement compréhensible que le langage machine
- il permet une plus grande portabilité, c'est-à-dire une plus grande facilité d'adaptation sur des machines de types différents

Un langage de haut niveau sert nous épargner la programmation en binaire. C'est pourquoi tout langage, à partir d'un programme écrit, doit obligatoirement procéder à une traduction en langage machine pour que ce programme soit exécutable.
Il existe deux stratégies de traduction, ces deux stratégies étant parfois disponibles au sein du même langage.
Le langage traduit les instructions au fur et à mesure qu'elles se présentent. Cela s'appelle la **compilation à la volée**, ou **l'interprétation**.
Le langage commence par traduire l'ensemble du programme en langage machine, constituant ainsi un deuxième programme (un deuxième fichier) distinct physiquement et logiquement du premier. Ensuite, et ensuite seulement, il exécute ce second programme. Cela s'appelle la **compilation**.

Il va de soi qu'un langage interprété est plus maniable : on peut exécuter directement son code au fur et à mesure qu'on le tape, sans passer à chaque fois par l'étape supplémentaire de la compilation. Mais il va aussi de soi qu'un programme compilé s'exécute beaucoup plus rapidement qu'un programme interprété : le gain est couramment d'un facteur 10, voire 20 ou plus.
Toute application destinée à un usage professionnel (logiciel par exemple) est forcément une application compilée.

Quelques exemples de langages couramment utilisés

Langage	Domaine d'application principal	Compilé / interprété
MAPLE	Calcul mathématique	langage interprété
MATLAB	Calcul mathématique	langage interprété
Mathematica	Calcul mathématique	langage interprété
BASIC	Comme son nom l'indique...	langage interprété
Prolog	Intelligence artificielle	langage interprété
Java	Programmation orientée internet	langage intermédiaire
LISP	Intelligence artificielle	langage intermédiaire
C	Programmation système	langage compilé
C++	Programmation système objet	langage compilé
Cobol	Gestion	langage compilé
Fortran	Calcul	langage compilé
Pascal	Enseignement	langage compilé

2. Le langage de programmation Maple

2.1 Généralité

2.1.1 Qu'est ce que Maple ?

Maple est un logiciel de mathématiques développé par des chercheurs de l'université de Waterloo au Canada et de l'université ETH de Zurich. Il se distingue par la puissance de son calcul symbolique, numérique et par la représentation graphique des résultats.
C'est un logiciel destiné aux scientifiques, ingénieurs, étudiants possédant un bon niveau mathématique.
Ce logiciel fait des merveilles dans le calcul à très haute précision, la résolution d'équations réelles, imaginaires, différentielles, intégrales, etc.

Le module graphique de base de Maple permet le changement de couleur discontinu et les changements dynamiques des points de vue de graphiques, Cette dernière caractéristique pouvant être très intéressante pour visualiser les surfaces et volumes en 3D.

Maple est un langage interprété, contrairement à **Pascal** et **Fortran** qui sont des langages compilés. Ça veut tout simplement dire que **Maple** est un interlocuteur toujours attentif et prêt à réagir à vos "paroles", sans passer par des étapes intermédiaires comme la compilation et l'établissement des liens (linking). On peut comparer les échanges avec un interpréteur à une conversation téléphonique, tandis que les compilateurs ressemblent davantage à un service postal: écrire la lettre, puis l'affranchir, puis... La conversation téléphonique se construit à

mesure, au fil des interactions et elle est largement improvisée. La lettre se compose en l'absence du destinataire et les réactions viendront... plus tard.
Bref, les interpréteurs sont des outils interactifs. Cette interactivité pousse rapidement à essayer des choses, juste pour voir ce que ça donne. Si vous faites une "erreur", elle ne pourra jamais être grave...même qu'elle pourrait très bien vous conduire sur des voies d'exploration que vous n'aviez même pas soupçonnées au départ.

D'autres produits analogues à Maple existent, notamment Derive, Mathematica, ou MuPad.
Derive est un produit beaucoup plus simple que Maple et bien moins puissant, mais il demande pour fonctionner des configurations moins évoluées que ses deux concurrents.
Mathematica est très puissant mais aussi très novateur : une syntaxe un peu contraignante, ainsi qu'une prise en main plus difficile le rendent moins facile à manipuler que Maple.
MuPad est développé par l'université de Paderborn en Allemagne. Il est maintenant muni d'une interface graphique développée en collaboration avec la société sciform. Ses performances sont comparables à celles de Maple et son prix est très attractif. On peut même en télécharger une version light gratuite.

2.1.2 Quelques généralités :

Une session de travail Maple commence évidemment par le lancement du logiciel en double cliquant sur l'icône correspondante. Une "feuille de travail" (worksheet) vide est alors affichée à l'écran, et Maple est prêt à travailler en mode interactif : on lui donne des instructions au clavier, que l'on valide par la touche Entrée, et il y répond de manière appropriée.
A tout moment, on peut fermer la feuille de travail, après avoir éventuellement sauvegardé son contenu.
On peut également charger une feuille de travail déjà sauvegardée, notamment celles qui accompagnent le logiciel et contenant des exemples d'utilisation.
On peut quitter Maple par l'une des instructions *quit, done, stop*. Le plus souvent cependant on sélectionne l'option *Exit* du menu *File*, ou avec la combinaison de touches *Alt+F4*.
Si on a modifié la feuille de travail depuis la dernière sauvegarde, Maple propose alors une sauvegarde ; on répondra par la négative pour les feuilles constituées d'exemples d'utilisation ou d'explications, et qui font partie du produit Maple.

Les connaissances de Maple ne sont chargées en mémoire qu'au fur et à mesure des besoins de l'utilisateur. Seule une faible partie, le noyau (kernel), est chargé dès le départ. La plus grande partie du savoir de Maple réside dans sa librairie (library). Quand l'utilisateur emploie, pour la première fois depuis le chargement du logiciel, une instruction figurant dans la librairie, les connaissances relatives à cette instruction sont chargées automatiquement en mémoire et y restent pour toute la durée de la session, sauf redémarrage par *restart*.
Un autre gisement de connaissances réside dans les packages, qui sont des regroupements de nouvelles fonctionnalités (fonctions, variables, textes d'aide et explications) liés à un domaine donné. Plusieurs packages font partie intégrante du logiciel, à titre d'exemple, voici certains modules avec le sujet traité:

Plots : Représentation graphiques
Linalg : Algèbre linéaire
Detools : Outils pour les Équations Différentielles
Powseries : Séries formelles.
Inttrans : Transformation intégrales : Laplace, Fourier.
Stats : Statistiques.

Pour la liste complète des modules, taper: *[> ?index, package;*

Si on veut, par exemple, travailler avec les matrices et les vecteurs, il est nécessaire d'introduire le module d'Algèbre linéaire du nom de `linalg` et cela, préférablement après la commande restart.

```
> restart;
  with (linalg);
```

Cette dernière commande affichera une série de commande pour traiter un grand nombre de notion d'Algèbre linéaire. Si on ne veut pas l'affichage de toutes les commandes, on remplace ; par *:* dans *with(linalg);*

On peut créer ses propres packages, mais cela demande une bonne connaissance du logiciel. Enfin on peut profiter des travaux réalisés par de nombreux utilisateurs expérimentés de Maple en chargeant en mémoire les packages ou feuilles de travail du répertoire *examples*.

2.1.3 Les icônes les plus utiles

Maple met à votre disposition plusieurs boutons vous permettant de lancer efficacement certaines des opérations non mathématiques les plus fréquentes. Ils sont tous très faciles d'utilisation et il vous suffira d'expérimenter un peu pour en apprendre rapidement l'usage. Nous nous contenterons donc d'attirer votre attention sur l'existence de certains de ces petits outils et d'émettre quelques brefs commentaires.

Bouton d'interruption: ***stop***

Vous voudrez parfois arrêter Maple en plein calcul parce que ça prend trop de temps ou parce que vous réalisez soudainement que vous lui avez demandé l'impossible. En ce cas, cliquez immédiatement sur l'icône *stop*.

Bouton de sauvegarde du document: ***Save***

La sauvegarde de documents (worksheet) est évidemment fondamentale. Pour effectuer cette opération, il vous suffira de dérouler le menu File, de faire glisser le curseur jusqu'à Save, puis de relâcher la souris. Un menu vous apparaîtra si vous sauvegardez ce document pour la première fois.

Bouton d'impression du document: ***Print***

Les documents Maple peuvent servir à construire du matériel imprimé (travaux, notes de cours,...). Le bouton d'impression Print se trouve dans le menu File et il est actionné de la même façon que le bouton de sauvegarde.

Autres

Vous observerez la présence de nombreux autres icônes et menus sur les barres d'outils. Plusieurs servent à "traiter du texte": copier, coller, effacer, changer le caractère, etc. Trois icônes situes au centre de la barre supérieure servent à créer des zones d'entrées de texte-commentaire ou de commandes Maple. Un bouton voisin (en forme de flèche pointant vers l'arrière) est utilisé comme touche d’effacement (DEL). Pour le reste, il vous suffira de cliquer ici et là et de voir ce que ça donne.

2.1.4 L’utilisation du help

L’utilisation de Maple demande de se conformer à une syntaxe assez stricte.
D’autre part il n’est pas facile de retenir le nom des centaines d’instructions disponibles, ainsi que la manière dont il faut les employer.

C’est pourquoi Maple est livré avec un système d’aide très complet.
Pour obtenir une aide sur une instruction dont on connaît le nom, comme par exemple expand, on peut procéder de plusieurs manières :
– *?expand* (suivi de Entrée) affiche l’écran d’aide complet sur cette instruction.
– *? ?expand* affiche la syntaxe : comment appeler expand, et avec quels paramètres.
– *? ? ?expand* affiche des exemples d’utilisation de l’instruction expand.
On peut également obtenir des aides sur des sujets entiers. Le mieux est alors de passer par le menu Help, ou d’utiliser la raccourcis-clavier indiquer dans ce menu :
– Browser (F1) : Un système hiérarchisé vous permet d’explorer toutes les possibilités du moteur algébrique de Maple, et en particulier de retrouver ou de découvrir les instructions qui correspondent à un sujet particulier.
– Interface Help (Shift+F1) : Ce système d’aide renseigne sur l’interface utilisateur (fenêtres, menus, icônes, utilisation de la souris, etc.)
– Key Word Search (Shift+F2) : Ce système vous permet d’obtenir une aide sur un sujet particulier ou plus général, à partir d’un mot que vous êtes invité à taper au clavier.
Maple affiche alors tous les sujets qui peuvent se rapporter à ce critère de recherche.
Il vous reste à choisir celui qui vous intéresse, mais attention : Maple parle anglais !
– Enfin Ctrl+F1 propose une aide contextuelle, c’est-à-dire relative à la position du curseur : si celui-ci se trouve sur le mot expand, l’écran d’aide correspondant est affiché.

2.1.5 Objets et Syntaxe sous Maple

Caractères spéciaux. Ils serviront à gérer votre conversation avec Maple. Ils ne font pas partie des commandes au sens strict. Le tableau suivant énumère ces "meta-caractères".

Qui ?	signe	signifie
Maple	>	Je suis à l'écoute
	syntax error	Je ne comprends pas
Vous	;	J'ai terminé et je veux que tu affiches la réponse
	:	J'ai terminé et je ne veux pas que tu affiches la réponse
	%	Prends le résultat précédent
	%%	Prends l'avant-dernier résultat
	? sujet	information à-propos de...
	#	J'inscris un commentaire

Constantes. Entre autres, il est capable de reconnaître des constantes de différents types :

Entier. Exemple : 5, 2,...

Fraction. Exemple : 1/5, 11/10, ...

Décimal. Exemple : 2., 3.65, ...

Symbolique. Exemple : Pi, I, E, infinity ...

des listes. Exemple : [1, 17, 5, 1]

des ensembles. Exemple : {1, 2, a}

Nombres différents. Pour Maple les nombres 1/5 et 0.2 ne sont pas identiques. Cette distinction peut vous paraître un peu ennuyeuse, mais elle vous permettra, entre autres choses, d'écrire exactement 1/3 et d'effectuer des calculs sans aucune erreur d'arrondi sur des entiers et des fractions.
Pourquoi ne pas écrire 22/7 plutôt que le symbole Pi? Parce que vous pouvez calculer tranquillement avec Pi ... sans erreur. Si, une fois vos calculs terminés, vous voulez numériser (décimaux), vous pouvez le faire avec toute la précision que vous désirez. Pour effectuer l'évaluation numérique il suffit d'employer la commande evalf.

Arithmétique. Maple connaît évidemment les opérations arithmétiques. Il interprète les +, -, / de la même façon que vous et moi

Fonctions mathématiques. Le nombre de fonctions mathématiques que Maple reconnaît est très considérable. En voici un tout petit échantillon : (sin, sqrt, exp, abs, ln, signum)

Calcul. Maple sait dériver, intégrer, prendre la limite, développer une fonction en série, résoudre des équations différentielles, etc.

Algèbre linéaire. Maple est particulièrement doué pour l'algèbre linéaire : addition, multiplication, inversion de matrices, produits scalaire et vectoriel, déterminant, transposée, résolution de systèmes, valeurs propres, base, espace des colonnes, orthogonalisation, ...

Bien Plus. Maple permet aussi de faire des statistiques, de la théorie des nombres, des graphes, et j'en passe beaucoup. En fait, le savoir mathématique de Maple est en constante évolution.

2.1.6 Quelques fonctions intéressantes de Maple :

sum	:	pour calculer des sommes
ifactor	:	pour factoriser en produit d'entiers
solve	:	pour résoudre une ou plusieurs équations
diff	:	pour dériver, différencier
dsolve	:	pour résoudre des équations différentielles
plot	:	pour tracer des courbes ou des solutions d'ED
fsolve	:	pour résoudre numériquement les équations
rsolve	:	pour résoudre des équations de récurrence
int	:	pour intégrer des fonctions
matrix	:	pour définir une matrice
eigenvals	:	pour calculer des valeurs propres
series	:	pour calculer la série de Taylor
limit	:	pour évaluer une limite

2.1.7 Commandes de base

Description	*Exemple*	*Sortie Maple*	*Commentaires*	*Note*
;	[> 3+4 ;	7	Exécute et affiche le résultat	
:	[> 3+4 :		Exécute la commande mais n'affiche pas le résultat	
:=	[> a := cos(5*Pi); [> b := a / 2.5;	a := -1 b:= -.4	C'est ce que l'on appelle une affectation	
evalf	[>A:=evalf(7/3); [>A:= evalf(7/3, 4);	A :=2.3333 A :=2.333	Évalue sous forme décimale Évalue avec 4 chiffres	
restart	[> restart;		re-initialise la feuille de travail	Se place au début de la feuille de travail.
simplify	[>simplify(cos(ß)^2 +sin(ß)^2);	1	Pour faire des simplifications	
rhs	[>eq:=3*sin(ß)=cos(ß): [> rhs(eq);	cos(ß)	rhs : abréviation du mot anglais right-hand- side	Permet de prendre le membre de droite d'une expression
lhs	[>eq:=3*sin(ß)=cos(ß): [> lhs(eq);	3 sin(ß)	lhs : abréviation du mot anglais left-hand- side	Permet de prendre le membre de gauche d'une expression
solve	[>solution:=solve(3*x=6, x);	solution:= x=2	Pour résoudre une équation	
int	[>res:=int(sin(x), x=-Pi..Pi);	res:= 0	Pour intégrer	
diff	[>dérivée:= diff(y(x),x,x);	dérivée:= $\frac{\partial^2}{\partial x^2} y(x)$	Dérivée seconde par rapport à x de y(x)	
subs	[>subs(x=2,sin(x*a));	sin(2 a)	Substitution de x=2 dans sin(x a)	
plot	[>plot(f(x), x=a..b):		Dessinera la fonction f(x) sur [a,b]	Commande valable en 2-D seulement
assume	[>assume(n,integer)		Sert à faire des hypothèses.	

2.1.8 Exercices

Exercice 2.1 :

1. Utilisation de `? <fonction>`, de `help` et de `restart`.
2. ouvrir une session Maple et effectuer les calculs suivants :

```
> 3+5;
> 3.5+4 :
> 3.5+4;
```

```
> 1/2+5/3;
> 1 : 2 : 3 : % ;
> 1 : 2 : 3 : %% ;
> 1 : 2 : 3 : %%% ;
> sin(0) ;
> Sin(0) ;
> 3**2 ;
  cos(Pi/4) ;
  ln(1) ;
> Pi ;
> evalf(%) ;
> I ;
> evalf(%^2) ;
> infinity ;
> % + 1 ;
```

Exercice 2.2 : exécuter les instruction suivantes :

1. `diff(x^2/(1+x^2),x) ;`
   `simplify(%) ;`
2. `sum(i^2,i=1..n) ;`
   `factor(%) ;`
3. `solve(a*x^2+b*x+c,x);`
   `factor(x^2-3*x+1) ;`

Exercice 2.3 :

1. calculer la dérivée de la fonction `f(x) = ln(x)/x^2`.
2. calculer la somme : $1^3+2^3+3^3+\ldots +n^3$.
3. résoudre les équations :
   - $x^2 – 3x +2 =0$,
   - $-x^2 + 4x + 3 = 0$.

Exercice 2.4 :

3. Evaluer le `sinus` de `(3π )/5`, en affichant 10 chiffres.
4. Evaluer `cos(ln(5)sin(3π/7)+5)`, en affichant 6 chiffres.

Exercice 2.5 : Exécuter les instructions suivantes :

```
> restart;
> a := 7;
> a := 0.5;
> b := 1;
> b := a;
> a; b;
> evalb(a=b);
> a := ’c’;
> a;
> 100!;
> length(%);
> whattype(%%);
> ifactor(100!);
> 42/5+3;
> whattype(%);
```

```
> numer(147/11);
> denom(147/11);
> evalf(147/11);
> whattype(%);
```

Exercice 2.6 : Soit le nombre complexe `z = (1+I)^2/(1-2*I).`

1. Calculer la partie réelle `Re(z)` et la partie imaginaire `Im(z)` de `z`.
2. Calculer le module de `z` : abs`(z)`.
3. Calculer l'argument de `z` : `argument (z)`.
4. Calculer le conjuguais de `z` : `conjugate(z)`.
5. Mettre z sous la forme `a + ib` en utilisant la fonction : `evalc()` .

Exercice 2.7 :

1. Stocker la constante `"infini"` ∞ dans la variable `x` et le nombre `ln(2)` dans la variable `y.`
2. Echanger les valeurs des variables `x` et `y.`
3. Vérifier les valeurs des variables `x` et `y` après l'échange.

Exercice 2.8 :

1. Exécuter les instructions suivantes (la commande `evalb`()):

```
> 4=5;
> evalb(4=5);
> evalb(1+3=4);
```

2. Exécuter les instructions suivantes (la commande `convert()`) :

```
> convert(123,binary);
> convert(100,hex);
> convert(101,decimal,binary);
> convert(`1A`,decimal,hex);
```

3. Exécuter les instructions suivantes :

```
> s :=a,b,c;
> s :=s,d;
> L :=[s];
> nops(L);
> op(L);
> op(2,L);
> L[3];
> op(1..2,L);
> L:=[op(L),e];
> E :={1,2,3,2};
> nops(E) ;
> op(E);
> op(2,E);
> convert([op(E)],`*`);
```

3. Maple et les Mathématiques

3.1 Les fonctions

3.1.1 Les fonctions mathématiques prédéfinies

Maple nous propose une foule de fonctions prédéfinies. Parmi celles-ci, on compte les ***sin***, ***log***, ***exp***, ***sqrt*** (racine carrée) que vous connaissez déjà très bien. En plus de ces grands classiques, d'autres relèvent de domaines spécialisés et beaucoup vous sont probablement inconnues. Elles suffiront dans la plupart des situations de calcul. Vous les rencontrerez en déroulant le menu **Help**, en cliquant sur **Contents** et en ouvrant la section **Mathematics**. Le reste est une question d'exploration et il y a beaucoup de choses.
Notez que toutes les fonctions "fonctionnent" de la même manière. Pour en tirer un calcul et un résultat, il vous suffira de taper le nom de la fonction correctement et de lui fournir un argument entre parenthèses:

```
> abs(-5);
```

5

Si vous connaissez le nom d'une fonction et que vous n'êtes pas certain(e) de ce qu'elle fait ou de la syntaxe de ses arguments, faites simplement :`?nom_de_fonction`
Par exemple, pour la fonction abs de "mise en valeur absolue":

```
>?abs
```

3.1.2 Définir des fonctions

Il est très commode de savoir définir ses propres fonctions pour tailler les calculs sur mesure. Les trois principaux mécanismes de définition sont:

- la notation fléchée ->
- l'opérateur unapply
- proc…end

notations fléchées

C'est certainement la façon la plus simple et elle rappelle la notation mathématique traditionnelle.
Voici, par exemple, une définition fléchée de la fonction "cosinus du carre".

```
> cos_carre :=x->cos(x^2);
```

cos_carre :=x->cos(x^2)

Une fois définie, elle acceptera divers types d'arguments:

```
>cos_carre(5) ;
```

cos(25)

```
>cos_carre(oiseau) ;
```

cos($oiseau^2$)

```
>cos_carre(1.5) ;
```

-.6281736227

avec *unapply*, à partir d'une expression

On peut avoir d'excellentes raisons de désirer construire une fonction à partir d'une expression. Le cas le plus typique est celui du polynôme dont on veut tirer une fonction polynomiale. Prenons par exemple:

```
>px :=55*x^5-37*x^4-35*x^3+97*x+50 ;
```

px := $-55\ x^5 - 37\ x^4 - 35\ x^3 + 97\ x + 50$;

On aimerait bien effectuer des évaluations en un point particulier de ***px***, en faisant simplement:

```
>px(3) ;
            -55 x(3)^5 - 37 x(3)^4 - 35 x (3)^3 + 97 x(3) + 50
```

On voit bien que ça ne marche pas du tout. La raison de ce comportement agaçant vient de ce que ***px*** n'est pas une fonction, mais une expression. Il faudra donc transformer ***px*** en une fonction. L'opérateur ***unapply*** sert précisément à cette transformation:

```
>fonc_px := unapply(px,x) ;
           fonc_px := x-> -55 x^5 - 37 x^4 - 35 x^3 + 97 x + 50
```

dès lors on obtient facilement

```
>fonc_px(3) ;
                                  -16966
```

avec *proc…end*

Les définitions de fonctions avec **proc…end** permettent d'écrire des "procédures", c'est-à-dire des séquences de commandes (à exécuter successivement). De plus, la programmation procédurale fait un usage fréquent de variables intermédiaires ainsi que d'instructions de contrôle (**if, for, while**).
Voici la définition d'une petite procédure. À partir d'une liste quelconque d'entiers, elle produit la sous-liste de tous les nombres premiers contenus dans cette liste.

```
>les_prem :=proc (liste_nombre)
           local nouvelle_liste, longueur, i ;
           nouvelle_liste := NULL ;
           longueur := nops(liste) ;
           for i from 1 to longueur do
                      if isprime(liste[i]) then
                           nouvelle_liste:=nouvelle_liste,liste[i] ;
                      fi
           od ;
           [nouvelle_liste] ;
end ;
liste := [seq(i,i=1..25)] ;
 liste := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
                   18, 19, 20, 21, 22, 23, 24, 25]
>les_prem(liste);
                  [2, 3, 5, 7, 11, 13, 17, 19, 23]
```

Cependant, une bonne connaissance des opérateurs fonctionnels comme **select** rend souvent superflue l'écriture de procédures. Dans le cas présent, on pourrait tout simplement écrire:

```
>liste_prem := (liste) -> select(isprime,liste) ;
            liste_prem := liste -> select(isprime,liste)
>liste_prem(liste) ;
                  [2, 3, 5, 7, 11, 13, 17, 19, 23]
```

3.2 Simplification

3.2.1 simplify

Une des activités mathématiques les plus importantes consiste à simplifier des expressions trop compliquées. Lorsque cette opération est bien réussie, une expression qui semblait

mystérieuse devient soudainement plus compréhensible. La simplification d'une expression permet aussi des gains en temps de calcul qui pourraient bien être considérables. A propos de simplification, il faut réaliser un certain nombre de choses lorsqu'on travaille avec **Maple**.
Maple effectue automatiquement certaines simplifications très élémentaires comme

(x -x) -> 0, x/x -> 1.

En dehors de ces cas évidents, **Maple** ne touche à rien à moins qu'on ne le lui dise.
La commande la plus facile à utiliser et qui donne généralement de bons résultats est **`simplify`**. Cette commande effectue des simplifications de toutes sortes et d'habitude les résultats concordent avec ce qu'on attend d'une simplification.
Cependant, il arrive que **`simplify`** ne simplifie pas beaucoup et même complique l'expression!
La notion de simplicité varie passablement en fonction des personnes et des situations.
Lorsque **`simplify`** ne peut pas faire le travail, il faut y aller avec des instructions plus précises. **Maple** en met plusieurs à notre disposition.

3.2.2 normal, collect, expand

`normal` donne une expression sous forme de fraction dite réduite, i.e. dont le numérateur et le dénominateur n'ont pas de facteur commun.

Exemple :
```
>normal(a/(a-b) – b*(a+b)/(a^2 – b^2));
                              1
```

`expand` brise les arguments d'une fonction, d'où des arguments plus simples, mais plus de termes.

Exemple:
```
>expand(sin(2*(a+b))) ;
                        4*sin(a)*cos(a)*cos(b)^2-
      2*sin(a)*cos(a)+4*cos(a)^2*sin(b)*cos(b)-2*sin(b)*cos(b)
```

`combine` en un sens, **`combine`** est l'inverse de **`expand`** : moins de morceaux, mais chacun est plus compliqué.

Exemple :
```
>combine(%,trig) ;
                        sin(2a + 2b)
```

`collect` ramasse les termes `` semblables ''.

Exemple :
```
>collect(7*(z+y)^3+x*(x+z*y)+(x+y)^2,y) ;
```
$$7\,y^3 + (21\,z + 1)\,y^2 + (21\,z^2 + 2\,x + xz)y + 7\,z^3 + 2\,x^2$$

`sort` met de l'ordre (les principes peuvent varier)

Exemple:
```
>poly :=expand((x-2)*(x^2+1)^3) ;
```
$$poly := x^7 + 3\,x^5 + 3\,x^3 + x - 2\,x^6 - 6\,x^4 - 6\,x^2 - 2$$

```
>sort(poly);
```
$$x^7 - 2\,x^6 + 3\,x^5 - 6\,x^4 + 3\,x^3 - 6\,x^2 + x - 2$$

factor effectue la mise en facteurs. C'est aussi une sorte d'inverse de la commande **expand.**

Exemple :
```
>factor(poly) ;
```
$$(x - 2)(x^2 + 1)^3$$

3.2.3 convert

La commande ***convert*** effectue deux genres de conversions:
i) d'un type de structure de données à un autre
ii) d'une forme d'expression à une autre
Nous donnons quelques exemples des deux aspects de cette commande extrêmement pratique.

Changements de type

Un usage très courant de cette commande consiste à faire l'addition ou la multiplication des éléments d'une liste par conversion du type **list** aux type **+** ou *****.

```
>liste := [1, 2, 3, 4, 5] ;
                          liste := [1, 2, 3, 4, 5]
>convert(liste, '+') ;
                               15
>convert(liste, '*') ;
                              120
```

Un autre usage répandu de la conversion de type consiste à transformer une série en un polynôme
```
>serie_tayl := taylor(sin(x),x) ;
          serie_tayl := x - 1/6 x^3 + 1/120 x^5 + O(x^6)
>convert(serie_tayl,polynom);
                  x - 1/6 x^3 + 1/120 x^5
```

Changements de forme

Donnons-nous l'expression trigonométrique suivante:
```
>express_trigo := sin(x)/cos(x) ;
```

On pourrait s'imaginer que la simplification automatique transformerait ce rapport en **tan(x)**
```
>simplify(express_trigo) ;
```
$$\frac{\sin(x)}{\cos(x)}$$

Non, il va falloir demander expressément à **Maple** d'effectuer cette "conversion":
```
>convert(express_trigo,tan) ;
                         tan(x)
```

La forme "fraction partielle" à partir d'une expression rationnelle est souvent recherchée. Aux yeux de **Maple**, il s'agit d'une conversion:
```
>convert((x^3+x-3)/(x^2-1),parfrac,x) ;
```

$$x - \frac{1}{2}\frac{1}{x-1} + \frac{5}{2}\frac{1}{x+1}$$

La forme de Horner des polynômes est particulièrement économique en nombre d'opérations et donc en temps de calcul. Cette forme s'obtient aisément grâce à la commande **convert** :

```
>poly := x^3+3*x^2+4*x-12 ;
                              poly := x^3 + 3x^2 + 4x - 12
>poly_horner:=convert(x^3+3*x^2+4*x-12,horner,x) ;
                    poly_horner := -12 + (4 + (3 + x) x) x
>expand(poly_horner);
                                   x^3 + 3 x^2 + 4 x - 12
```

3.3 Extremums d'une fonction

Trois commandes de base sont disponibles pour trouver les extremums d'une fonction d'une ou de plusieurs variables :
Extrema (expression, {contrainte}, {variable})
Minimize (expression, option1, option2, …, option3)
Maximize (expression, option1, option2, …, option3)

Exemple : Trouver le minimum de $(x-2)^2$ sur l'intervalle [1,5]

```
>restart ;
 fonction := (x-2)^2 ;
 minimize(fonction, x=1..5,location) ;
```

location indiquera quelles sont les coordonnées du point où se trouve ce minimum

Exemple : trouver le minimum et le maximum de f(x,y)=x2+y2 sur le domaine [-3,3]x[-4,4]

```
>restart ;
 fonction := x^2 + y^2 ;
 minimize(fonction, x = -3..3, y = -4..4, location) ;
 maximize(fonction, x = -3..3, y = -4..4, location) ;
```

Exemple : soit une sphère centrée à l'origine de rayon 2 et dont l'équation est x2+y2+z2=2. Sachant que la température sur la sphère est donnée par la fonction :
$T(x, y, z) = x^2 + y^2 + z^2$, trouver les extremums de T(x, y, z).

```
>restart ;
 fonction := x^2 + y^2 + z^2 ;
 extrema (fonction, x^2+y^2+z^2=2, {x, y, z}) ;
```

3.4 Graphisme

Grâce aux commandes **plot** et **plot3d**, vous pourrez produire des graphes de plusieurs fonctions dans une même fenêtre, des graphes paramétrés, etc. Nous en rencontrerons quelques exemples et le **?** vous en fournira plusieurs autres.
Les commandes graphiques **plot** et **plot3d** sont les plus fréquemment utilisées. Le package **plots** en contient plusieurs autres. Pour obtenir des informations sur ce package : faites simplement **?plots**. Tapez **with(plots):** pour le charger et ainsi avoir accès à ces commandes supplémentaires. Notez la présence du petit **:** à la fin de cette dernière instruction. C'est que **Maple** est particulièrement bavard lorsqu'il charge un package.

3.4.1 Graphisme 2-D

Pour dessiner des courbes en deux dimensions, plusieurs commandes sont disponibles. Le choix de l'une de ces commandes se fait selon que la fonction est exprimée sous forme : explicite, implicite, paramétrique ou sous forme polaire.

Forme explicite

soit à dessiner la fonction f = f(x) sur l'intervalle [a,b].
plot(f(x), x=a..b) ;

Exemple : dessiner la fonction f(x) = sin(x) sur [p, 4p]

```
>plot(sin(x), x = Pi .. 4*Pi);
```

Exemple : Dessiner sur un meme graphique les fonctions f(x) et g(x) ci-dessous, dont f(x) sera en bleu et g(x) en rouge, sur l'intervalle [-p, p]
f(x) = sin(x)
g(x) = cos(x)

```
>plot([sin(x),cos(x)], x = -Pi..Pi, color = [blue,red]);
```

Remarque: [sin(x),cos(x)] étant une liste de même que [blue, red] ainsi l'ordre des fonctions à dessiner suivra l'ordre de la liste des couleurs.

Forme implicite

Soit à dessiner la fonction qui est donnée sous la forme implicite f(x, y) = 0.
Pour cela, il est nécessaire d'introduire le module **plots**. Ainsi :

```
>with(plots) ;
 implicitplot( f(x, y), x = a .. b, y = c .. d) ;
```

Exemple : Dessiner la courbe dont l'équation cartésienne est :
Exp(xy) = cos(x-y)　　　　　　pour $0<x<\pi$ et $2<y<4$
Noter que y = y(x).

```
>with(plots) ;
implicitplot(exp(x*y) – cos(x – y), x = 0 .. Pi, y = 2 .. 4) ;
```

Forme paramétrique

Maple permet aussi de représenter des courbes définies paramétriquement, c'est-à-dire dont les composantes x et y dépendent toutes deux d'un même paramètre.
Soit à tracer une fonction sur l'intervalle [a,b] et dont les équations paramétriques sont :
x = f(t)
y = g(t)
la syntaxe de la commande est :

```
plot ([f(t), g(t), t = a .. b], scaling = constrained) ;
```

Exemple : soit à tracer un cercle de rayon R = 2 sur l'intervalle [0 .. 2p], dont les équations paramétriques sont :
x(t) = 2 * cos (t)
y(t) = 2 * sin(t)

```
>plot([2*cos(t), 2*sin(t), t = 0 .. 2*Pi], scaling = constrained) ;
```

Forme polaire

Soit à tracer la cardioïde dont l'équation polaire est r(β) = 1 + cos(β), sur [0, 2p], dans le système de coordonnées cartésiennes.

```
>with(plots) ;
 r := 1 + cos(beta) ;
 polarplot(r, beta = 0 .. 2*Pi) ;
```

3.4.2 Graphisme 3D

Forme explicite

Soit à dessiner une surface dont l'équation de cette surface est donnée sous forme z=f(x,y).
La syntaxe de la commande est :

```
>plot3d ( f(x,y), x = a .. b, y = c .. d) ;
```

Exemple : soit à dessiner la surface $x^2 + y^2 = z$ pour $-1 < x < 1$ et $-2 < y < 2$

```
>plot3d( x^2 + y^2, x= -1..1, y=-2..2) ;
```

Forme implicite

Soit à dessiner la surface S lorsque l'équation de cette surface est donnée sous forme : f(x,y,z)=0.
Les commandes sont :

```
>with(plots) ;
implicitplot3d ( f(x,y,z), x=a..b, y=c..d, z=e..f) ;
```

forme paramétrique

Soit à dessiner une courbe dans l'espace dont on connaît les équations paramétriques suivantes :

$x = f(t)$
$y = g(t)$ $\quad b < t < a$
$z = h(t)$

les commandes sont :

```
>with(plots) ;
spacecurve([f(t),g(t),h(t)], t=a..b) ;
```

<u>**Exemple :**</u> soit à tracer dans l'espace l'hélice circulaire dont les équations paramétriques sont :

$x = \cos(t)$
$y = \sin(t)$ $\quad 0 < t < 2\pi$
$z = t$

les commandes sont :

```
>with(plots) ;
spacecurve([cos(t),sin(t),t], t = 0..2*Pi) ;
```

soit à dessiner une surface dont on connaît les équations paramétriques suivantes :

$x = f(t,s)$
$y = g(t,s)$ $\quad a < t < b$ et $c < s < d$
$z = h(t,s)$

la commande est :

```
>plot3d([f(t,s),g(t,s),h(t,s)], t = a..b, s = c..d) ;
```

3.5 Dérivation

3.5.1 Dérivation explicite

La syntaxe est :
Diff(expression, x1, x2, … xn);

Exemple : soit à calculer ∂f(x,y)/∂x où f(x,y) = sin(x+y) ln(xy)

```
>expression := sin(x +y) * ln(x*y) ;
  diff(expression,x) ;
```

Exemple : soit à calculer $\partial^2 f(x,y)/\partial x \partial y$ où f(x,y) = sin(x+y) ln(xy)

```
>expression := sin(x +y) * ln(x*y) ;
   diff(expression, y, x) ;
```

3.5.2 Dérivation implicite

La syntaxe est :
implicitdiff(équation, fonction dépendante, variable de dérivation) ;

Exemple : soit à trouver dy/dx de cos(x+y) = ln(xy) où y=y(x).

```
>expression := cos(x+y) = ln(x*y) ;
  implicitdiff(expression, y, x) ;
```

3.6 Intégration

3.6.1 Intégration simple

La syntaxe est :
int(f(x), x = a..b) ; pour évaluer l'intégrale et afficher le résultat.
Int(f(x), x = a..b) ; pour retourner l'intégrale non évaluée.

Tel que :
- f(x) est la fonction à intégrer
- x est la variable d'intégration
- a,b sont les bornes d'intégration inférieure et supérieure respectivement

Exemple : soit à intégrer la fonction f(x) = sin(x)

```
>int(sin(x),x) ;
```

Soit à intégrer la fonction f(x) = sin(x) sur [-p, p]

```
>int(sin(x), x = -Pi .. Pi) ;
```

3.6.2 Intégrale double

La syntaxe est :
Doubleint(f(x,y), x, y) l'intégrale se fera selon x et ensuite selon y
Doubleint(f(x,y), x = a..b, y = c..d)
Doubleint(f(x,y), x, y, domaine)
Tels que :
- f(x,y) est la fonction ou l'expression à intégrer.
- a, b, c, d sont les bornes d'intégration.
- Domaine est le nom du domaine qui apparaîtra sous les intégrales

Pour utiliser cette commande, il est nécessaire d'ouvrir le module *student* à l'aide de `with(student)`. Voir exemple ci-dessous.

Exemple : soit à intégrer la fonction f(x,y) = x+y sur le domaine D=[1,2]x[3,4]

```
>with(student) ;
  doubleint(x+y, x = 1..2, y = 3..4) ;
  evalf(%) ;                            le signe % fait référence au dernier résultat
```

3.7 Développement en série de Taylor

3.7.1 Série de Taylor

La syntaxe est :

taylor (f(x), x = a, n) ;

tels que:

- f(x) est la fonction dont il faut écrire le développement en série de Taylor.
- a est le point autour duquel f(x) est développé en série de Taylor
- n est l'ordre du développement

Exemple : soit à développer en série de Taylor autour du point x=0 de f(x)=sin(x).

```
>serie_taylor := taylor(sin(x), x = 0,3) ;
                      serie_taylor := x + O(x^3)
```

3.7.2 Polynôme de Taylor

De la série de Taylor, il est possible d'extraire le polynôme de Taylor à l'aide de la commande `convert` et de l'option `polynom`.

Exemple :

```
>serie_taylor := taylor(sin(x), x = 0,3) ;
                      serie_taylor := x + O(x^3)
>poly := convert(serie_taylor, polynom);
                              poly := x
```

Pour les séries entières, on peut utiliser le module `powseries` qui contient plusieurs commandes permettant de manipuler ces séries.

```
>restart ;
  with(powseries) ;
```

3.8 Exercices

Exercice 3.1 :

1. Donner les termes de la suite x^i pour `i` variant de `-5` à `7` (utiliser la commande `seq`).
2. Calculer le produit des termes de la suite x^i pour `i` variant de `-5` à `7` (utiliser la commande `convert`).

Exercice 3.2 :

1. Mettre dans une liste les valeurs du polynôme $x^2 + x + 41$ pour `x` entier variant de `-10` à `10`, (on définit le polynôme par `p :=x->x^2+x+41`).
2. Sélectionner les éléments de cette liste qui sont des nombres `premiers`. Que constatez-vous ? (utiliser la fonction `isprime()`).

Exercice 3.3 : Exécuter les instructions suivantes :

```
1. > for n from 5 to 10 do
         convert((n-1)^2,base,n);
         convert(2*(n-1),base,n) ;
     od;

2. > for n from 5 to 10 do
         convert((n-1)^3,base,n);
         convert((n+2)*(n-1)^2,base,n) ;
     od;
3. Que constatez-vous ?
```

Exercice 3.4 :

Étant donnés deux matrices d'ordre 3 à coefficients réels suivantes :
`A :=matrix(3,3,[-1,2,0,4,-2,3,0,1,-3])` et
`B :=matrix(3,3,[2,0,1,4,-1,1,2,0,-5])`.

1. Calculer la somme des deux matrices A et B,
2. Calculer le produit du nombre `-5` par la matrice A,
3. Calculer la trace de la matrice A et celle de la matrice B,
4. Calculer le produit de la matrice A par la matrice B,
5. Calculer les déterminants `det(A)` et `det(B)`.

Exercice 3.5 :

On note `H_a_b:=matrix(3,3,[a,b,b,b,a,b,b,b,a]),` la matrice d'ordre 3 à coefficients réels.

1. Calculer la trace de la matrice `H_a_b`, pour quelles valeurs de a et b la trace de `H_a_b` est toujours nulle ?
2. Calculer le produit matricielle `H_a_b` × `H_c_d`.
3. Calculer le déterminant de la matrice `H_a_b`.
4. On se place dans le cas de `a=1`, pour quelles valeurs de `b` la matrice `H_1_b` est-elle inversible ?
5. On se place dans le cas de `b=1`, pour quelles valeurs de `a` la matrice `H_a_1` est-elle inversible ?

4. Eléments de programmation

4.1 Les tests

Il arrive souvent que la décision d'effectuer telle ou telle tâche dépende de la réalisation d'une condition, on a alors recours au ***if***.

4.1.1 L'instruction conditionnelle simple

La syntaxe est la suivante :

```
if condition then
     I1
     I2
     …
     Ik
fi;
```

Explication : si la condition est vraie exécuter les instructions $I_1, \ldots, I_k$.

Exemple :

```
> a:=10;b:=25;
  if (a<=b) then
       print(b);
  fi;
  if (a>b) then
       print(a);
  fi;
```

$$a := 10$$

$$b := 25$$

$$25$$

Pour éviter la répétition des instructions if on utilise l'instruction conditionnelle avec else.

4.1.2 L'instruction conditionnelle avec else

La syntaxe est la suivante :

```
if condition then
     I1
     …
     Ik
else
     I'1
     …
     I'k
fi;
```

Explication : si la condition est vraie exécuter les instructions $I_1, \ldots, I_k$ sinon exécuter les instructions $I'_1, \ldots, I'_k$.

Exemple :

```
> a:=10;b:=25;
```

```
if (a<b) then
      print(b);
else
      print(a);
fi;
```

$a := 10$

$b := 25$

25

4.1.3 L'instruction conditionnelle imbriquée

La syntaxe est la suivante :

```
if cond1 then
      Instr1
elif cond2 then
      Instr2
      …
elif condk-1 then
      Instrk-1
else
      Instrk
fi;
```

Explication : si la condition $cond_1$ est vraie exécuter les instructions $Instr_1$, sinon, si la condition $cond_2$ est vraie exécuter les instructions $Instr_2$, sinon …, sinon, si la condition $cond_{k-1}$ est vraie exécuter les instructions $Instr_{k-1}$ sinon exécuter $Instr_k$.

Exemple :

```
> a:=1; b:=1; c:=1; delta:=b*b-4*a*c;
  if (delta>0) then
        print("deux solutions réelles");
  elif (delta=0) then
        print("une solution double réelle");
  else
        print("deux solutions complexes");
  fi;
```

$a := 1$

$b := 1$

$c := 1$

$\delta := -3$

"deux solutions complexes"

4.2 Les opérateurs logiques and, or et not (ET, OU et NON)

Lorsqu'on souhaite vérifier plusieurs conditions simultanément, on a recours aux opérateurs logiques.

and --> vrai si condition1 et condition2 sont satisfaites

or --> vrai si condition1 ou condition2 est satisfaite

not --> vrai si condition n'est pas satisfaite

Exemple: On veut savoir à quelle partie du domaine D = [-5,8] appartient x.
Si (x < -5) et (x < 8) alors
x est à l'intérieur du domaine
sinon ((x=-5) ou (x = 8)) alors
x est à l'extérieur du domaine
sinon
x est sur la frontière du domaine
fin si

```
>x := 9;
                          x := 9
>if (x > -5 and x < 8) then
                printf(`x extérieur`)
            elif not (x = -5 or x = 8) then
                printf(`x extérieur`)
            else
                print(`x frontière`)
 fi
                                x extérieur
```

4.3 Les itérations

Les commandes itératives servent à répéter une séquence d'instructions convenablement modifiées à chaque étape. Elles sont présentes dans **Maple** sous cinq formes : les boucles ***for … do*** et ***while*** qui existent dans plusieurs langages de programmation, ainsi que les commandes ***seq, $*** et ***map***.

4.3.1 La boucle for

La boucle ***for*** sert à répéter la même opération plusieurs fois en faisant varier un indice d'itérations. Une syntaxe non exhaustive de la boucle for est la suivante :

```
for<variable>from<début>by<saut>to<fin>do<instructions> end do
```

Les commandes ***from*** et ***by*** sont optionnelles, elles ont une valeur par défaut de 1. les autres commandes sont obligatoires.

Exemple :

```
>for i from 4 to 30 by 7 do
                print(i)
 od;
                         4     11     18     25

>restart;
l:=[a,b,c,d,e];
for i in l do
          print(i)
od;
                       a     b     c     d     e
```

L'instruction `break` permet de sortir de la boucle.
L'instruction `next` permet de passer à l'itération suivante dans la boucle.

Exemple :
```
> for i from 1 to 10 do
      if i=4 then next;fi;
      if i=6 then break; fi;
     print(i);
od;
                                 1     2     3     5     6
```

4.3.2. La boucle while

Il arrive souvent qu'on veuille répéter un groupe d'instructions jusqu'à ce qu'une certaine condition soit satisfaite. C'est justement le type de contrôle que permet d'exercer la commande ***while***, sa structure syntaxique est la suivante :

while <condition> do <instruction> end do

Exemples :

1.
```
>i:=4:
    while i<=30 do
        print(i);
        i:=i+7;
    od:        # : au lieu de ; pour ne pas afficher le i
                   4     11     18     25
```

2.
```
>restart; l:=[a,b,c,d,e]: i:=1:
    while i<=nops(l) do
        print(l[i]);
        i:=i+1;
    od:        # : au lieu de ; pour ne pas afficher le i
                 a     b     c     d     e
```

4.3.3 La commande seq

Si f(i) est le terme général d'une suite, alors la commande

seq(f(i),i=m..n)

génère la suite : f(m), f(m+1), …, f(n).

cette commande peut être utilisée pour calculer des sommes ou des produits. Voici un autre exemple

Exemple : produit cartésien -- voici la suite des couples d'un produit cartésien d'ensembles A*B.

```
>restart :
  A := {a,b,c,d} :
  B := {1,2,3}:
  Seq(seq([i,j],i=A),j=B) ;
   [a,1],[b,1],[c,1],[d,1],[a,2],[b,2],[c,2],[d,2],[a,3],[b,3],[
                            c,3],[d,3]
```

4.3.4 La commande $

La commande ***$***, tout comme la commande ***seq***, est utilisée lorsqu'on désire construire une suite ***f(i)*** avec un paramètre i variant entre a et b. la syntaxe de la commande ***$*** est la suivante :

'f(i)'$'i' = a .. b

Exemple : construisons la suite $1^2, 2^2, 3^2, 4^2, 5^2$ avec la commande $

```
>'i^2'$'i'=1..5 ;
                      1, 4, 9, 16, 25
```

4.3.5 La commande map

La commande ***map(f,L)***, utilise également l'itération pour appliquer une procédure ***f*** aux membres d'une liste ***L*** ou, de façon plus générale, aux opérandes d'une expression.

Exemples :

1 . `>map(D, [x ->x^2, x ->x^3, sin]) ;`

$$[x \to 2x, x \to 3x^2, \cos]$$

2. `>f := x -> log[10](x) ;`

$$f := \log_{10}$$

`>map (f, [1,10,100,1000]) ;`

$$\left[0, 1, \frac{\ln(100)}{\ln(10)}, \frac{\ln(1000)}{\ln(10)}\right]$$

`>simplify(%) ;`

$$[0, 1, 2, 3]$$

```
> f := (i,j) -> x^((i+1)^j) ;
> A := matrix(2,2,f);
```

$$A := \begin{bmatrix} x^2 & x^4 \\ x^3 & x^9 \end{bmatrix}$$

`> map(diff,A,x);`

$$\begin{bmatrix} 2x & 4x^3 \\ 3x^2 & 9x^8 \end{bmatrix}$$

4.4 Procédures

Les programmes, ou procédures, utilisent la commande fondamentale `proc`. Ecrivons, dans un premier exemple, la procédure pour obtenir la partie entière de la moitié d'un nombre n.

Exemple : partie entière de n/2.

```
> moitie := proc(n)
   trunc(n/2) ;
   end proc ;
            moitie := proc(n)trunc(1/2*n) end proc
> moitie(5);
                          2
```

La forme syntaxique générale de la commande proc est la suivante:

```
proc(x1 :: type1, x2 :: type2, x3 :: type3, …)
```

```
         local v1, v2, v3, …. ;
         global u1, u2, u3, … ;
         options op1, op2, … ;
         iinstructions
     end proc ;
```

seules les commandes `proc` et `end` proc sont obligatoires dans une procédure. Toutes les autres instructions sont facultatives. Les paramètres x1, x2, … sont appelés les arguments ou les entrées de la procédure. Dans les lignes suivantes nous faisons la description des composantes d'une procédure.

4.4.1 Les types des arguments dans une procédure

Les arguments d'une procédure appartiennent souvent à un type spécifique. On peut insérer pour chaque argument `xi` une contrainte `typei` sur son type en ajoutant celle-ci immédiatement après `xi` dans la première ligne de la procédure avec la syntaxe `xi :: typei.` Par exemple, si on veut contraindre l'entrée n à être un nombre dans la procédure `moitie`, on écrit :

```
> moitie := proc(n :: numeric)
   trunc(n/2) ;
  end proc ;
       moitie := proc(n :: numeric)trunk(1/2*n) end proc
> moitie(a);
Error, invalid input : moitie expects its 1st argument, n, to be
                of type numeric, but received a
```

4.4.2 Variables locales et variables globales

L'instruction `local` dans une procédure `f` fournit la liste des variables dont la portée est limitée à la procédure `f`. Redéfinissant la procédure `moitie` pour donner à la variable `b` la valeur `n/2`, mais seulement à l'intérieur de la procédure :

```
> b :=     100 ;
moitie := proc(n)
local b ;
b := n/2 ;
trunc(b) ;
end proc ;
  moitie := proc(n)local b ;b :=1/2*n ; trunc(b) end proc
> moitie(5) ;
                              2
> b ;
                             100
```

Comme on le voit, la variable b a une valeur globale de 100 et une valeur locale à l'intérieur de la procédure moitie de n/2.

La commande `global` donne la liste des variables qui ont la même valeur à l'intérieur et à l'extérieur d'une procédure.

```
> c := 70 ;
   b := 100 ;
   moitie := proc(n)
              local b ; global c ;
         b := trunc(n/2) ; c := 374 ;
         print(b) ; print(c) ;
end proc :
```

```
> moitie (5) ;
                                        2
                                       374
> b ;
                                       100
> c ;
                                       374
```

Exemples :

1. # procédure qui donne le factoriel de n

```
> # procédure qui calcule la somme des entiers de 1 à n

> somme := proc(n)    # ici pas de point virgule
      local s,i;          # variables locales
      s := 0;
     for i from 1 to n do
          s := s + i;
     od:
     s;
end;
```

$somme := \textbf{proc}(n)\ \textbf{local}\ s, i; s := 0; \textbf{for}\ i\ \textbf{to}\ n\ \textbf{do}\ s := s + i\ \textbf{end do}\ ; s\ \textbf{end proc}$

```
> somme(10);
```

55

```
> somme(100);
```

5050

2. # procédure qui donne le factoriel de n

```
restart;
facto := proc(n)
     local p,i;
     p:=1;
     for i from 1 to n do
          p := p*i;
     od;
     p;
end;
```

$facto := \textbf{proc}(n)\ \textbf{local}\ p, i; p := 1; \textbf{for}\ i\ \textbf{to}\ n\ \textbf{do}\ p := p \times i\ \textbf{end do}\ ; p\ \textbf{end proc}$

```
> facto(5);facto(10);
```

120

3628800

```
> t:=time(): facto(300): time()-t;
```

.001

```
> t:=time(): facto(30000): time()-t;
```

17.356

3. La fonction en Maple qui donne le nombre de bits nécessaire pour coder un entier n en binaire est :

```
> nbits := proc(n)
     local nb, i;
     i:=iquo(n,2);
```

```
        nb:=1;
        while(i<>0) do
             i:=iquo(i,2); /* le quotient de i par 2 */
             nb:=nb + 1;
        od;
        nb;
     end;
```

- La fonction Maple iquo(n,k) renvoie le résultat de la division entière (quotient) de n par k.
  exemple : iquo(7,2) donne 3.
- L'expression n mod k donne le reste de la division de n par k.
  exemple : 7 mod 2 donne 1.

4.4.3 Valeur de retour d'une procédure

La valeur renvoyée par une procédure `f` est le résultat de la dernière évaluation faite dans la procédure. Les commandes `RETURN` et `ERROR` permettent de modifier cette règle. La commande `ERROR(message)` interrompt l'exécution de la procédure et affiche le message fourni en interdisant toute valeur de retour de la procédure.

Exemple :

```
> moitie := proc(n)
   if not(type(n,numeric)) then
     ERROR('l'argument doit être de type numérique')
  else trunc(n/2) ;
  end if ;
  end proc ;
> moitie(x) ;
   ERROR, (in moitie) l'argument doit être de type numérique
```

La commande `RETURN(x)` insérée dans une procédure termine l'exécution de la procédure et désigne x comme valeur de retour.

Exemple : Appartenance à une liste – la procédure suivante vérifie si une expression x appartient à une liste L.

```
> element := proc(x :: anything, L :: list)
     local i;
     for i to nops(L) do
          if L[i]=x then RETURN(vrai) end if;
     end do;
     faux;
end proc;
>element(a,[a,b,e,a]);
                              Vrai
>element(c,[a,b,e,a]);
                              Faux
```

4.4.4 Procédures récursives

Une procédure récursive est une procédure qui s'appelle elle-même.

Exemple : calcul du factoriel d'un nombre entier

```
> facteur := proc(n)
     local f;
```

```
      if n<>1 then
            f:=n*facteur(n-1);
      else return(n);
      end if;
> end proc;
```

facteur := **proc**(*n*) **local** *f*; **if** *n* <> 1 **then** *f* := *n***facteur*(*n* - 1) **else** **return** *n* **end if**; **end proc**;

```
> facteur(3);
```

6

4.4.5 Sauvegarde de procédures et de calculs

Lorsqu'on veut conserver des procédures écrites lors d'une séance de travail ou encore le résultat d'un long calcul, il est possible de sauvegarder ce travail en format interne ***Maple*** pour le réutiliser dans une séance de travail ultérieure. La commande ***save*** sert à copier du texte dans un fichier dont le nom se termine par ***.m***. la syntaxe de cette commande est :
Save listenoms,fichier.m ;

Exemple : on veut sauvegarder la procédure facteur de l'exemple précédent dans un fichier appelé fichierfact ainsi que le résultat de facteur(5). On fait :

```
> n := facteur(5) :
        save facteur,n,"fichierfact.m ";
```

Maple ne renvoie aucune réponse mais un fichier `fichierfact.m` a alors été crée et on peut retrouver le contenu de ce fichier lors d'une séance ultérieure avec la commande read.

```
> restart ;
   n ;
   read " fichierfact.m_";
   n;
                                        120
```

4.5 Exercices

Exercice 4.1 :

4. Étant donnés deux nombres `a` et `b`, afficher le plus grand des deux nombres.

5. Soit la fonction `f(x)=(x+4)/(x^2-1)`,
   a. Calculer, f(2) et f(1),
   b. Ecrire une suite d'instructions qui affiche un message pour dans le cas de la division par zéro.

6. Soit l'équation `ax^2+bx+c=0`, les nombres `a`, `b` et `c` sont donnés :
   a. Calculer `delta`,
   b. Afficher le type et le nombre de solution que possède cette l'équation.

Exercice 4.2 :

On sait que l'état de l'eau dépend de sa température. Écrire une procédure qui affiche l'état de l'eau (solide, liquide ou gazeux) selon sa température.

Exercice 4.3 :

Etant donnés deux nombres a et b, écrire une procédure qui nous informe si le signe du produit de a et b est négatif, positif ou nul (**attention :** on ne doit pas calculer le produit des deux nombres).

Exercice 4.4 :
Écrivez une procédure qui permet de discerner une mention à un étudiant selon la moyenne de ses notes :

- "Très bien" pour une moyenne comprise entre 16 et 20,
- "Bien" pour une moyenne comprise entre 14 et 16,
- "Assez bien" pour une moyenne comprise entre 12 et 14,
- "Passable" pour une moyenne comprise entre 10 et 12.

Exercice 4.5 :

3. Afficher les entiers compris entre -10 et 20. Mettre les entiers compris entre -10 et 20 dans une liste.
4. Afficher les entiers impairs compris entre 115 et 231, ensuite mettez les dans une liste.
5. Stocker les nombres premiers inférieur à 100 dans une liste.

Exercice 4.6 :

1. Écrivez un algorithme qui calcule la somme des `n` premiers nombres entiers positifs. Vérifier le résultat en utilisant `sum()`.
2. Écrivez un algorithme qui calcule le factoriel de `n`, où `n` est un entier positif. Vérifier le résultat en utilisant le factoriel : `n!`

Exercice 4.7 :

1. Ecrire une procédure en Maple qui calcule la trace d'une matrice carrée d'ordre n.
2. Ecrire une procédure en Maple qui calcule la transposée d'une matrice carrée d'ordre n.
3. Ecrire une procédure en Maple qui calcule la somme de deux matrices carrées d'ordre n.
4. Ecrire une procédure en Maple qui calcule le produit d'une constante par une matrice carrée d'ordre n.
5. Ecrire une procédure en Maple qui calcule le produit de deux matrices carrées d'ordre n.
6. Ecrire une procédure en Maple qui calcule le déterminant d'une matrice d'ordre 3 à coefficients réels.

Exercice 4.8 :
Refaire les exercices 2.5 et 2.6 en utilisant les résultats de l'exercice 3.7.

Exercice 4.9 : La suite de `Fibonacci` est donnée par les équations :

$$\begin{cases} u_0 = 0 \\ u_1 = 1 \\ u_n = u_{n-1} + u_{n-2} \quad \text{pour} \quad n \geq 2 \end{cases}$$

1. Calculer u_i pour `i` variant de `2` à `7`.
2. Calculer u_i pour `i` variant de `2` à `50` (utiliser la boucle `for`).

Exercice 4.10 :

1. Ecrire un programme qui donne le plus grand élément d'une liste L donnée.
2. Ecrire un programme qui calcule la valeur décimale d'un entier représenté en binaire par une liste de 0 et 1.
   Vérifier le résultat en utilisant la fonction `convert()`.

Exercice 4.11 :

Recherche d'un élément dans une liste de nombres : recherche séquentielle

1. Ecrire une procédure qui prend pour arguments une liste `L` et un nombre `x`, et elle retourne `true` ou `false` selon que `x` appartient ou non à la liste L.
2. Améliorer la procédure précédente pour que la recherche s'arrête dès qu'on a trouvé la valeur `x`.

Exercice 4.12 :

Le tri d'une liste de nombres : réarranger les éléments d'une liste dans l'ordre croissant pour rendre plus efficaces les opérations de recherche et, par conséquent, d'insertion, de suppression, etc.

1. Tri par sélection :
   Principe :
   - ranger le plus petit élément,
   - trier le reste de la liste.

   Ecrire une procédure Tri_selection() : qui prend pour argument une liste L, et elle ordonne la liste L dans l'ordre croissant selon le principe de tri par sélection.

2. Tri par insertion :
   Principe : Cette méthode est très utilisée lorsqu'on joue aux cartes. Les éléments (les cartes) sont divisés en une suite destination $a_1 \ldots a_{i-1}$ et une suite source $a_i \ldots a_n$. A chaque étape, en partant de `i=2` et en augmentant `i` de 1, on prend le $i^{ème}$ élément de la suite source et on l'insère à sa place dans la suite destination. Pour insérer l'élément couramment considéré, on déplace simplement les élément qui lui sont supérieurs un cran vers la droite et on l'insère dans la place laissée vacante.

   Ecrire une procédure Tri_insertion() : qui prend pour argument une liste L, et elle ordonne la liste L dans l'ordre croissant selon le principe de tri par insertion.

3. On peut vérifier le résultat du tri en utilisant la fonction `sort()` de Maple.

Exercice 4.13 :

Recherche d'un élément dans une liste : recherche dichotomique
On se place dans le cas où la liste L est ordonnée et les éléments de la liste sont deux à deux distincts.
Principe : Nous cherchons à savoir si une valeur x est présente dans la liste L. Pour cela, nous comparons `x` à l'élément `L[mil]` situé au milieu de la liste. Si `x>=L[mil]`, alors `x` est dans la partie de la liste à droite de `mil`. Sinon, il est dans la partie située à gauche. Dans les deux cas, le nombre d'emplacements possibles pour `x` a été divisé par deux.

1. Ecrire une procédure `recherche_dicho()`, qui s'inspire du principe dichotomique précédent.
2. Comparer la recherche dichotomique à la recherche séquentielle.

5. Exercices et Problèmes d'examens

5.1 Examen d'informatique (I_2 SMP-SMC) session de Juin 2004

Enoncés des exercices :

Exercice 1 :

Ecrivez un algorithme qui demande à l'utilisateur d'entrer la température de l'eau, et affiche ensuite l'état de l'eau selon la température (on rappelle que l'état de l'eau est glace pour une température inférieure ou égale à 0°, est vapeur pour une température supérieure ou égale à 100° et liquide pour une température comprise strictement entre 0° et 100°).

Exercice 2 :

1. Ecrivez un algorithme qui lit un entier n et compte le nombre de 1 dans la représentation binaire de cet entier.
2. Écrivez un algorithme qui lit un tableau d'entiers de n éléments et donne la plus grande et la plus petite valeur de ce tableau.

Exercice 3 :

Soit la fonction mystere écrite en Maple suivante :

```
mystere := proc(n)
     local liste,x,i ;
     x := n ;
     liste := NULL ;
     i :=2 ;
     while (i<= x) do
          if (irem(x,i)=0) then
               x := x/i;
               liste := liste,i;
          else
               i := i+1;
          fi;
     od;
     [liste];
end;
```

1. Que valent `mystere(8)`, `mystere(90)` et `mystere(210)` (justifier par un tableau de variables) ?
2. Que fait ce programme ?
3. Que se passera-t-il si on remplace l'instruction `i:=2;` par `i:=1;` ?

N.B : la fonction Maple `irem(x,y)` retourne le reste de la division de `x` par `y` et `x/y` désigne la division entière de `x` par `y`.

Exercice 4 :

Deux nombres entiers `n` et `m` sont qualifiés d'**amis**, si la somme des diviseurs de `n` est égale à `m` et la somme des diviseurs de `m` est égale à `n` (on ne compte pas comme diviseur le nombre lui même et 1).

Exemple : les nombres 48 et 75 sont deux nombres **amis** puisque :

- Les diviseurs de 48 sont : `2, 3, 4, 6, 8, 12, 16, 24` et
  `2 + 3 + 4 + 6 + 8 + 12 + 16 + 24 = 75`
- Les diviseurs de 75 sont : `3, 5, 15, 25` et `3 + 5 + 15 + 25 = 48`.

Ecrire un algorithme qui permet de déterminer si deux entiers `n` et `m` sont amis ou non.

5.2 Correction de l'examen d'informatique (I_2 SMP-SMC) session Juin 2004

Exercice 1 : sur 4 points

1ère solution :

```
Variable Temp : Entier
Début
     Ecrire("Entrez la température de l'eau :" )
     Lire(Temp)
     Si Temp =< 0  Alors
          Ecrire("C'est de la glace" )
     Sinon  Si Temp < 100 Alors
               Ecrire("C'est du liquide")
            Sinon
               Ecrire("C'est de la vapeur")
Fin
```

2ème solution (mauvaise):

```
Variable Temp : Entier
Début
     Ecrire("Entrez la température de l'eau : ")
     Lire(Temp)
     Si Temp =< 0 Alors
          Ecrire("C'est de la glace")
     Si Temp > 0 Et Temp < 100 Alors
          Ecrire("C'est du liquide")
     Si Temp > 100 Alors
          Ecrire("C'est de la vapeur")
Fin
```

Exercice 2 : sur 7 points

1.

```
Variables i,n,poids : entiers
Debut
     Ecrire(" Entrer la valeur de n :")
     lire(n)
     i ← n
     poids ←  0
     TantQue(i<>0) faire
          si (i mod 2 = 1) alors
               poids ← poids + 1

          i ← i/2
     FinTantQue
     Ecrire(poids)
```

```
        Fin
```

2.
```
variables Tableau Tab[100], i, n, min, max : Entier
debut
     ecrire("donner la taille du tableau :")
     lire(n)
     ecrire("donner les éléments du tableaux :")
     Pour i allant de 1 à n faire
          lire(Tab(i))
     FinPour
     min ← Tab[1]
     max ← Tab[1]
     Pour i allant de 2 à n faire
          si (min > Tab[i]) alors
               min ← Tab[i]
          si (max < Tab[i]) alors
               max ← Tab[i]
     FinPour
     Ecrire("le minimum est ",min)
     Ecrire("le maximum est ",max)
fin
```

Exercice 3 : sur 6.5 points

1. `mystere(8) :`

x	8	4	2	1
i	2	2	2	2
liste		2	2,2	2,2,2

`mystere(90) :`

x	90	45	45	15	5	5	5	1
i	2	2	3	3	3	4	5	5
liste		2	2	2,3	2,3,3	2,3,3	2,3,3	2,3,3,5

`mystere(210) :`

x	210	105	105	35	35	35	7	7	7	1
i	2	2	3	3	4	5	5	6	7	7
liste		2	2	2,3	2,3	2,3	2,3,5	2,3,5	2,3,5	2,3,5,7

2. Ce programme donne la liste des diviseurs (premiers) d'un entier n.

3. En remplaçant l'instruction `i :=2` par l'instruction `i :=1` le teste `irem(x,i)=0` est toujours vrai, `x` prend toujours la même valeur et la liste contiendra une infinité de `1` (on obtient **une boucle infini**) !

Exercice 4 : sur 3 points

1ère Solution :

```
variables n, m, d, p, S1, S2 : Entier
 debut
       ecrire(" entrer l'entier n :")
```

```
        lire(n)
        ecrire(" entrer l'entier m :")
        lire(m)
        S1 ← 0
        S2 ← 0
        d ← 2
        TantQue d*d<=n faire
             Si irem(n,d)=0 alors
                  S1 ← S1 + d + n/d
             d ← d + 1
        FinTantQue
        p ← 2
        TantQue p*p<=m faire
             Si irem(m,p)=0 alors
                  S2 ← S2 + p + m/p
             p ← p + 1
        FinTantQue
        Si (S1=S2) alors
             Ecrire(" n et m sont amis")
        Sinon
             Ecrire(" n et m ne sont pas amis")
   fin
```

2ème Solution :

```
  variables n, m, i, S1, S2 : Entier
   debut
        ecrire(" entrer l'entier n :")
        lire(n)
        ecrire(" entrer l'entier m :")
        lire(m)
        S1 ← 0
        S2 ← 0
        Pour i allant de 2 à n-1 faire
             Si irem(n,i)=0 alors
                  S1 ← S1 + i
        FinPour
        Pour i allant de 2 à m-1 faire
             Si irem(m,i)=0 alors
                  S2 ← S2 + i
        FinPour
        Si (S1=S2) alors
             Ecrire("n et m sont amis")
        Sinon
             Ecrire("n et m ne sont pas amis")
   fin
```

5.3 Examen d'informatique (I_2 SMP-SMC) session de Juillet 2004

Exercice 1 : (4 points)

Les étudiants ayant passé l'examen du module I2 en session de Juin ont été classés selon leurs notes en trois catégories :

- pour une note inférieure strictement à 5, l'étudiant est éliminé,
- pour une note supérieure ou égale à 5 et inférieur strictement à 10, l'étudiant passe la session de rattrapage,
- pour une note supérieure ou égale à 10, l'étudiant valide le module

Ecrivez un algorithme qui demande à l'utilisateur d'entrer la note du module, puis affiche la situation de l'étudiant selon sa note (on suppose que l'utilisateur entre une note valide entre 0 et 20).

Exercice 2 : (4 points: 1/3)

Un nombre **parfait** est un entier positif supérieur à 1, égal à la somme de ses diviseurs ; on ne compte pas comme diviseur le nombre lui-même.
Exemple : 6 est un nombre parfait puisque : 6 = 3 + 2 + 1.

1. Donner un nombre parfait différent de 6.
2. Ecrire la conception de l'algorithme qui nous dit si un entier n est parfait ou non.

Exercice 3 : (5 points: 3/2)

Soit la fonction mystere écrite en Maple suivante :

```
mystere := proc(a,b)
     local r ;
     while ( b > 0 ) do
          r := irem(a,b);
          a := b;
          b := r;
     od;
     a;
end;
```

4. Que valent `mystere(35,12)`, `mystere(96,81)` et `mystere(34,21)` (donner les valeurs que prennent les variables a, b et r dans chacun des cas) ?
5. Que fait ce programme ?

N.B : la fonction Maple `irem(x,y)` retourne le reste de la division de `x` par `y`.

Exercice 4 : (7 points: 3/4)

1. Écrivez un algorithme qui lit la taille d'un tableau n, le tableau T, une valeur x, et il indique ensuite si l'élément x appartient ou non au tableau T.

2. Écrivez un algorithme qui permet de déterminer si les éléments d'un tableau d'entiers sont tous consécutifs ou non. (Par exemple, si le tableau est : 7; 8; 9; 10, ses éléments sont tous consécutifs. Si le tableau est : 7; 9 ; 10; 11, ses éléments ne sont pas tous consécutifs).

5.4 Correction de l'examen d'informatique (I_2 SMP-SMC) session juillet 2004

Exercice 1 :

1ère version :

```
Variables note : réel
Début
   Ecrire("Entrez la note du module :")
   Lire (note)
   Si (note < 5)   alors
        Ecrire ("l'étudiant est éliminé")
   Sinon Si note <10 alors
              Ecrire ("l'étudiant passe en rattrapage")
        Sinon
              Ecrire ("l'étudiant a validé le module")
        Finsi
  Finsi
Fin
```

2ème version :

```
Variables note : réel
Début
   Ecrire("Entrez la note du module")
   Lire (note)
   Si (note < 5)   alors
        Ecrire("l'étudiant est éliminé")
    Finsi
   Si (note >= 5 et note < 10) alors
        Ecrire(" l'étudiant passe en rattrapage")
   Finsi
   Si (note > 10) alors
        Ecrire("l'étudiant a validé le module")
  Finsi
Fin
```

Exercice 2 :

1. Le nombre 28 est parfait puisque 28 = 14+4+7+2+1.
2. **1ère solution** :

```
Variables n, d, S : entier
Début
     Ecrire ("Entrez la valeur de n : ")
     Lire (n)
     S ← 1
     d ← 2
     TantQue (d*d <= n)
          Si (n%d=0) alors
               S ← S + d + n/d
          d ← d + 1
     FinTantQue
     Si (S=n) alors
          Ecrire ("le nombre n est parfait ")
```

```
    Sinon
        Ecrire ("le nombre n n'est pas parfait ")
Fin
```

2ème solution :

```
Variables n, d, S : entier
Début
    Ecrire ("Entrez la valeur de n : ")
    Lire (n)
    S ← 0
    d ← 1
    TantQue (d < n)
        Si (n%d=0) alors
            S ← S + d
    FinTantQue
    Si (S=n) alors
        Ecrire ("le nombre n est parfait ")
    Sinon
        Ecrire ("le nombre n n'est pas parfait ")
Fin
```

Exercice 3 :

1. `mystere(35,12) :`

a	35	12	11
b	12	11	1
r	11	1	0

La valeur affichée est **1**

`mystere(96,81) :`

a	96	81	15	6
b	81	15	6	3
r	15	6	3	0

La valeur affichée est **3**

`mystere(34,21) :`

a	34	21	13	8	5	3	2
b	21	13	8	5	3	2	1
r	13	8	5	3	2	1	0

La valeur affichée est **1**

2. Cet algorithme donne le plus grand commun diviseur (le pgcd) de deux nombres entiers a et b.

Exercice 4 : (**7 points: 3/4**)

1.

variables n, i, x, T[20] : **entiers**
debut
    ecrire("entrer la valeur de n : ")
    lire(n)
    ecrire("entrer le tableau T : ")
    pour i allant de 1 à n faire
        lire(T[i])

```
        finpour
        ecrire("entrer la valeur de x : ")
        lire(x)
        i ← 1
        TantQue ( i ≤ n  et  T[i] ≠ x) faire
                i ← i + 1
        FinTantQue
        Si (i ≠ n+1) alors
                Ecrire(" l'élément x se trouve dans le tableau T ")
        Sinon
                Ecrire(" l'élément x ne se trouve pas dans le tableau T ")
        finsi
  fin
```

2. Dans cette question, on suppose que les variables n et T sont connues.

1ère version:

```
Variable i: entier
Début
  i ← 1
  TantQue (i<n ET T[i+1] = T[i]+1)
     i ← i+1
  FinTantQue
  Si (i<n) alors
    Ecrire("les termes du tableau ne sont pas consécutifs")
  Sinon
    Ecrire(" les termes du tableau sont consécutifs ")
  FinSi
Fin
```

2ème version:

```
Variable i, j: entier
Début
   j ← 1
   Pour i allant de 1 à n-1
        Si (T[i+1] ≠ T[i] +1) alors
                j ← 0
                i ← n-1
        FinSi
   FinPour
   Si (j=1) alors
    Ecrire("les termes du tableau sont consécutifs")
  Sinon
    Ecrire("les termes du tableau ne sont pas consécutifs")
  Finsi
Fin
```

5.5 Examen d'informatique (I_2 SMP-SMC) session de Juin 2005

Exercice 1 :
Ecrire un algorithme qui demande l'âge d'un enfant à l'utilisateur. Ensuite, il l'informe de sa catégorie :

- La catégorie d'un enfant est "Poussin" si 6 ≤ age < 8,
- La catégorie d'un enfant est "Pupille" si 8 ≤ age < 10,
- La catégorie d'un enfant est "Minime" si 10 ≤ age < 12,
- La catégorie d'un enfant est "Cadet" si age ≥ 12.

Exercice 2 :

3. Ecrivez un algorithme qui lit un entier n et compte le nombre de 1 dans la représentation binaire de cet entier.
4. Écrivez un algorithme qui lit un entier n (la taille du tableau), le tableau d'entiers T de n éléments, l'entier x et indique le nombre de fois que x figure dans le tableau T.
5. Ecrivez une procédure qui prend pour arguments un entier n (la taille du tableau), le tableau de réels T et affiche le plus grand élément du tableau T ainsi que sa position dans le tableau. (On suppose que le tableau T est formé d'éléments tous distincts)

Exercice 3 :
Soit la fonction mystere écrite en Maple suivante :

```
>   mystere := proc(L,t)
     local u, i;
     u := 0;
     for i from 1 to nops(L) do
          u := u*t + L[i];
     od;
     u;
end;
```

6. Que valent `mystere([1,-1,0,2],3)`, `mystere([1,0,1,1,1],2)` et `mystere([a,b,c],x)` (donner la valeur de `u` à chaque étapes de la boucle) ?
7. Que fait ce programme ?
8. Que calcule la procédure `mystere(L,t)` si les éléments de la liste `L` sont tous strictement inférieur à `t` ?

Exercice 4 :
Soit `T` un tableau qui contient `n` valeurs réelles triés dans l'ordre croissant. Ecrire une procédure qui prend comme paramètre le Tableau `T`, l'entier `n` (la taille de `T`) et un nombre réel `x`, et elle effectue l'insertion de `x` dans le tableau `T`, de telle manière que le tableau `T` reste trié.

Exemple : Soit le tableau `T` de `8` nombres triés dans lequel on désire insérer le nombre `40` :

4	7	8	12	23	56	89	112

Le résultat est un tableau `T` de 9 nombres toujours triés :

4	7	8	12	23	40	56	89	112

5.6 Examen d'informatique (I_2 SMP-SMC) session de Juillet 2005

Exercice Exercice 1 : (4pts)
Écrivez un algorithme qui effectue la lecture de la moyenne d'un étudiant et affiche sa mention sachant que :
- la mention "Très bien" est décernée pour une moyenne comprise entre 16 et 20 ($16 \leq$ moyenne ≤ 20)
- la mention "Bien" est décernée pour une moyenne comprise entre 14 et 16 ($14 \leq$ moyenne < 16)
- la mention "Assez bien" est décernée pour une moyenne comprise entre 12 et 14 ($12 \leq$ moyenne < 14)
- la mention "Passable" est décernée pour une moyenne comprise entre 10 et 12 ($10 \leq$ moyenne < 12)

Exercice 2 : (7pts = 3pts + 4pts)
1. Un nombre entier p (différent de 1) est dit premier si ses seuls diviseurs positifs sont 1 et p. Ecrivez un algorithme qui effectue la lecture d'un entier `p` et détermine si cet entier est premier ou non.
2. Écrivez un algorithme qui lit la taille d'un tableau `n`, le tableau `T`, une valeur `x`, et il indique ensuite si l'élément `x` appartient ou non au tableau `T`.

Exercice 3 : (5pts = 4pts + 1pts)
Soit la fonction `mystere` écrite en Maple suivante :

```
mystere := proc(a,b)
     local r ;
     while ( b > 0 ) do
          r := irem(a,b); # le reste de la division de a par b #
          a := b;
          b := r;
     od;
     a;
end;
```

1. Que valent `mystere(35,15)`, `mystere(132,81)` et `mystere(55,34)` (donner les valeurs que prennent les variables `a`, `b` et `r` dans chacun des cas) ?
2. Que fait ce programme ?

Exercice 4 : (4pts)
Ecrivez une procédure qui prend pour paramètres l'entier `n` et le tableau de réels `T`, et qui affiche le produit du plus petit élément du tableau T avec le plus grand élément du tableau T.

Exemple : Si `T` est le tableau suivant :

-4	0	6	1	3	5	-8	2

Le résultat affiché sera `-8×6=-48.`